Proposed Standard

A Structured Error Handler

Background

This proposal is derived from the informal discussion held at the end of the September standards conference on error handling. It is designed to meet the requirement for an error handler that is structured, can be nested easily.

Two implementations of error handlers existed at the time of the last conference, Unicomal and RC. This proposal follows the general outline of the RC handler (since this met with more approval at the meeting), although it incorporates some features from the Unicomal version. The Unicomal handler works along very different lines to those proposed.

Proposal

The user can specify that when an error occurs, execution will transfer to a given procedure. This is done with the statement

ERROR <procedure identifier>

Where <procedure identifier> specifies a procedure that can be found by the system. It may be desirable to allow error handlers to be in library files. The only limitation on the procedure specified is that it may not have any parameters. When the statement is encountered, it should check that no parameter list is present (even if this involves accessing a library) and generate an error if not - this error would of course find its handler through the previous contents of the error vector.

The default error handler can be restored by the statement

ERROR

without a procedure name.

When a error procedure has been declared as active, any error occuring will cause the current location to be saved and the procedure to be executed. The procedure is a normal procedure without any parameters. When the procedure is entered, two things happen depending on whether it is open or CLOSED. If it is open then the error vector is restored to the default and its old value is lost. If it is CLOSED then the old error vector (i.e. pointing to the procedure that is being entered) is saved and then the default vector is specified. Within the error procedure, a new error vector may be declared (which can be the current one). If the procedure is open, the effect of this change lasts even after the error procedure is executed. If it is CLOSED then the previous error vector is restored on exit.

At any point dynamically inside the error handling procedure, the following statements may be executed:

RETRY Re-attempts the simple statement that was executing when the error occurred.

IGNORE Resumes execution at the simple statement after the one which caused the error (or the next iteration of the same simple statement if the error occurred in a short form loop). This statement would not be allowed after certain types of fatal errors, for instance "No value RETURNed from function". In such cases an ABORT would occur with an extra message, for instance:

```
IGNORE not allowed after
Error at line 2340
No value RETURNed from function
2340 ENDFUNC
```

ABORT Causes the default error handler to report the error. It then drops into command level.

Any of the above commands will cause a run-time error if they do not occur dynamically inside an error handling sequence, for instance if the procedure is called normally. If the error handling procedure performs none of these and runs through to the ENDPROC statement, then an ABORT will be forced.

To facilitate handling of errors, several functions will be provided:

ERRNUM Returns the number of the last error. Reset to 0.

ERR$ Returns the message corresponding to the last error. Reset to "No error" or the equivalent in another language (preferably the one native the machine's users)

ERRLINE Returns the number of the line containing the error. Reset to 0.

Possible extra functions (though these are not part of the proposal) are:

ERRLINE$ A string containing a "listing" of the line on which the error occurred. Reset to "No error".

ERRPTR An offset pointing to the location of the error in the above string. Reset to 0.

The values of these functions are all copied onto the stack when an error handling routine is entered, before they are set up to the values relevant to the error being handled. They are restored when the procedure is exited via IGNORE or RETRY. These functions are reset to the values given as defaults by the commands CLEAR, RUN and NEW.

*** End of Proposal ***

All of the above is probably best explained with examples:

```
100 ERROR black
110 a:=0
120 b:=1/a
```

```
130 PRINT b
140 END
150 //
160 PROC black
170   CASE ERR$ OF
180   WHEN "Division by zero"
190     RETRY
200   OTHERWISE
210     ABORT
220   ENDCASE
230 ENDPROC
```

This will cause the follwing steps to occur:

100 Set the error handling vector to point to procedure black.
110 Assign value to global variable a.
120 Evaluate 1/a as 1/0. Generate a division by zero error. Save location of error, and all necessary information to retry or ignore it. Transfer execution pointer to the start of procedure black. Reset the error vector to point to the default error handler. Set up the values of ERRLINE, ERRNUM and ERR$. Continue execution from line 170.
170 Evaluate ERR$, find that it is "Division by zero". Continue execution from line 190.
190 RETRY restores the pointers to the simple statement, and resumes execution.
120 Again a division by zero error occurs: save location of error, etc., and since the error vector points to the default error handler, output error message and stop.

This is obviously a badly written error handler, since RETRYing a division by zero error will almost always cause a recurrence of the error. However, it does underline one of the reasons for setting the error vector to point to the default error handler when entering an error procedure. Another, perhaps more important, reason for doing this would be if procedure black contained another error, for instance if it tried to open a file to read a more detailed error message and that file could not be found. As a more trivial example, consider:

```
100 ERROR white
110 x=1/0
120 y=1/x
130 PRINT y
140 END
150 //
160 PROC white
170   PRINT "Value of y when error occurred was "¡y
180   ABORT
190 ENDPROC
```

In this case the error handler was designed to handle a potential error at line 120. When the error at 110 occurs, the value of y is undefined and so an "Undefined variable" error occurs at line 170. If the error vector still pointed to procedure white at this point it would go into an infinite loop re-entering the procedure white.

Sometimes it is desireable to be able to handle an error and have the same facility to handle the next one. This can be done in two ways. If it is necessary to modify global variables from within the error handler, then an open error handler can be used which, before it re-enters execution of the main program points the error vector back at itself. For instance

```
100 PROC red
110   errorcount:+1
120   PRINT "Ignoring ";ERR$
130   ERROR red
140   IGNORE
150 ENDPROC
```

The other way of doing this involves using a CLOSED procedure. When this is left the error vector will automatically be restored to its previous value. For instance:

```
100 PROC blue CLOSED
110   IF ERR$ = "Escape" THEN ABORT
120   RETRY
130 ENDPROC
```

This can also access global variables if it employs an IMPORT statement.

Error handlers can be nested. For instance:

```
100 PROC main_handler
110   ERROR sub_error
120   handle:=1
130   record:=ERRNUM
140   OPEN FILE handle,"Alternative_messages",RANDOM 50 READ ONLY
150   INPUT FILE handle,record:mess$
160   PRINT mess$;" at ";ERRLINE
170   CLOSE FILE handle
180   STOP
190 ENDPROC
200 //
210 PROC sub_error
220   IF ERR$="File already open" THEN
230     handle:+1
240     RETRY
250   ELSE
260     PRINT "Original error number ";record
270     PRINT "Cannot find alternative error messages"
280     ABORT
290   ENDIF
300 ENDPROC
```

Note that the ERRLINE accessed at line 160 corresponds to the error causing main_handler to be entered, since it is preserved through any call to sub_error.

Problems with the proposed method

One major problem is apparent. When the error occurs in the course

of obtaining a value, e.g. "Division by zero", then if the error is IGNOREd, what value should be used. For instance in the first example if line 190 said IGNORE instead of retry, would value of b should be output in the subsequent PRINT statement? There does not seem to be a logically consistent value that can be chosen for a wide variety of such errors. In view of this, it would probably be wise to add a powerful extension to the error handling facilities. The extra complications that this involves are a much lesser problem than the one just outlined.

Additional Proposal

The ERROR statement may take a function as an argument. This will behave exactly like the procedures specified above, but will enable on extra command within it:

RETURN Will return a value to replace one that could not be evaluated. If this is executed when an error has occurred for which it is not applicable, then an error will occur. Errors which will allow RETURN should include "Undefined variable", "Division by zero", "Log range", "-ve SQR", "no value RETURNed by function" and other similar errors. In these cases it must be specifically stated to what point the value is returned. This will be very clear to implementors, but may not be so apparent to users.

It may be preferable to specify that only procedures may be used as error handlers, but allow the RETURN statement within the procedure.